API Documentation

Contents

Intro

VR360Player is an open source DirectX-based application that renders 360° videos on Oculus Rift. We achieve this goal efficiently by utilizing hardware decoding accelerators on the latest Intel processors (gen. 6+). The player is built on top of Media SDK's decode sample, this manual merely deals with the relevant changes and additions, for further documentation please refer to the official manuals.

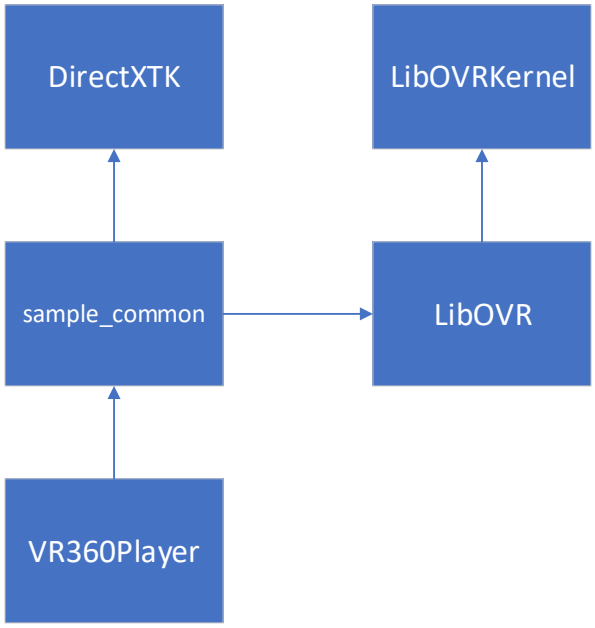
Prerequisites

Building the solution requires some libraries and SDKs to be installed and set up.

1. Windows 10 SDK with MFC and ATL support
2. Intel Media SDK with environment variable INTELMEDIASDKROOT pointing to the SDK root.
3. mfx_vs2015.lib with debug symbols (build from source at INTELMEDIASDKROOT\opensource).
4. ffmpeg-win64-dev and ffmpeg-win64-shared with environment variable FFMPEG_ROOT pointing to *dev* directory. *shared* directory must be included in PATH.
5. [optional] Intel VTune Amplifier with environment variable VTUNE_AMPLIFIER_2018_DIR pointing to VTune directory. To cancel VTune integration, define INTEL_NO_ITTNOTIFTY_API macro in VR360Player and sample_common preprocessor definitions.

Solution File Structure

The solution consists of 5 dependent projects. The projects and their dependencies are depicted in a dependency graph:



DirectXTK – DirectX Tool Kit is a library maintained by Microsoft that offers common classes and structures for DirectX11 applications. The library offers an efficient sphere that we use as the player's ambient.

LibOVRKernel and LibOVR – Oculus SDK libraries used to manage the session with the HMD.

sample_common – The common classes and helper methods offered by Intel Media SDK. We introduce a new class to this library that supports d3d11 rendering on Oculus Rift.

VR360Player – The executable project. It statically links to all the libraries above and provides FFMPEG integration, a decode-render loop and motion control. Built on top of Media SDK's decode sample.

Command-Line Switches

VR360Player is built on top of Intel Media SDK's decode sample. Therefore, all the switches that existed on the decode sample still apply. We will focus on the switch that deals with OVR rendering.

General structure:

<codec id> -ovr -i InputFile

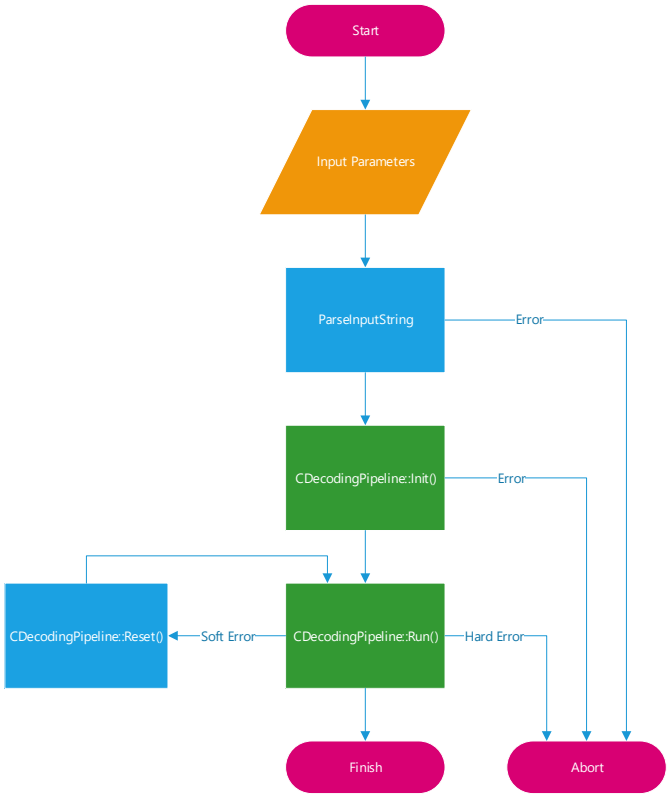
<codec id> = h264|h265|mpeg2 (tested and working)

-ovr = Oculus VR Rendering mode. Using this switch adjusts the proper configurations needed for Oculus. It also adjusts the player's frame delivering rate to the native rate encoded in the input file headers.

InputFile = Video file name (video stream must be encoded with the codec specified above)

VR360Player Overview

An overview of the player workflow, for the main executable module is depicted in the image below.



The input parameters are passed to the player via command-line switches, once they are parsed and validated, an initialization process begins. Any error found on the steps mentioned leads the player to abort. Finally, the decode-render loop begins execution, aborting on hard errors and resetting if possible. The green rectangles represent the two main parts of the player: the initialization and the decode-render loop, we will deal with these in more detail later.

CHWDevice Interface

Belongs to sample_common. To understand the API and eventually modify it, we explain it using a bottom-up approach. This interface poses a contract that must be fulfilled by new classes if they wish to introduce different implementations such as WinMR/HTC Vive/etc.

Most of the following methods are self-explained, clarifications are given for the rest.

```
class CHWDevice
{
public:
    virtual ~CHWDevice(){}
    /** Initializes device for requested processing.
    @param[in] hWindow Window handle to bundle device to.
    @param[in] srcFPS frame rate of input file.
    */
    virtual mfxStatus Init(mfxHDL hWindow, size_t srcFPS) = 0;
    virtual mfxStatus Reset() = 0;
    // Get handle can be used for MFX session SetHandle calls
    virtual mfxStatus GetHandle(mfxHandleType type, mfxHDL *pHdl) = 0;
    /** Set handle.
    Particular device implementation may require other objects to operate.
    */
    virtual mfxStatus SetHandle(mfxHandleType type, mfxHDL hdl) = 0;
    virtual mfxStatus RenderFrame(mfxFrameSurface1 * pSurface, mfxFrameAllocator *
pmfxAlloc) = 0;
    virtual void      Close() = 0;
};
```

CD3D11DeviceOVR

Belongs to sample_common. This is the class responsible for rendering frames to the Oculus Rift. It fulfills the above contract using D3D11 APIs and the Oculus SDK that manages the HMD session.

The Init method saves the window handle and discards the other parameters since they are irrelevant in the class implementation. The method then proceeds to initialize LibOVR and HMD session. A D3D11 Device and Context are created and initialized and so are the texture set, depth buffer, render target view and a viewport for each eye. A sphere is deployed with the camera in the center and finally a mirror texture is created to render the final frame on the screen as well. The swap chain and the back buffer are created and initialized in the Reset method which is called after a proper window handle is passed with a SetHandle call initiated by the CDecodeD3DRender instance.

The RenderFrame method extracts a ID3D11Texture2D pointer from the surface and allocator parameters and proceeds to calculate the camera position and direction based on the HMD sensors and keyboard input. Then the texture is applied to the inner-sphere and rendered to each of the eyes, the final frame is eventually copied to the mirror window.

The class is furthered extended with an OnKey() method that can change the camera location and facing direction based on key-strokes.

CDecodeD3DRender

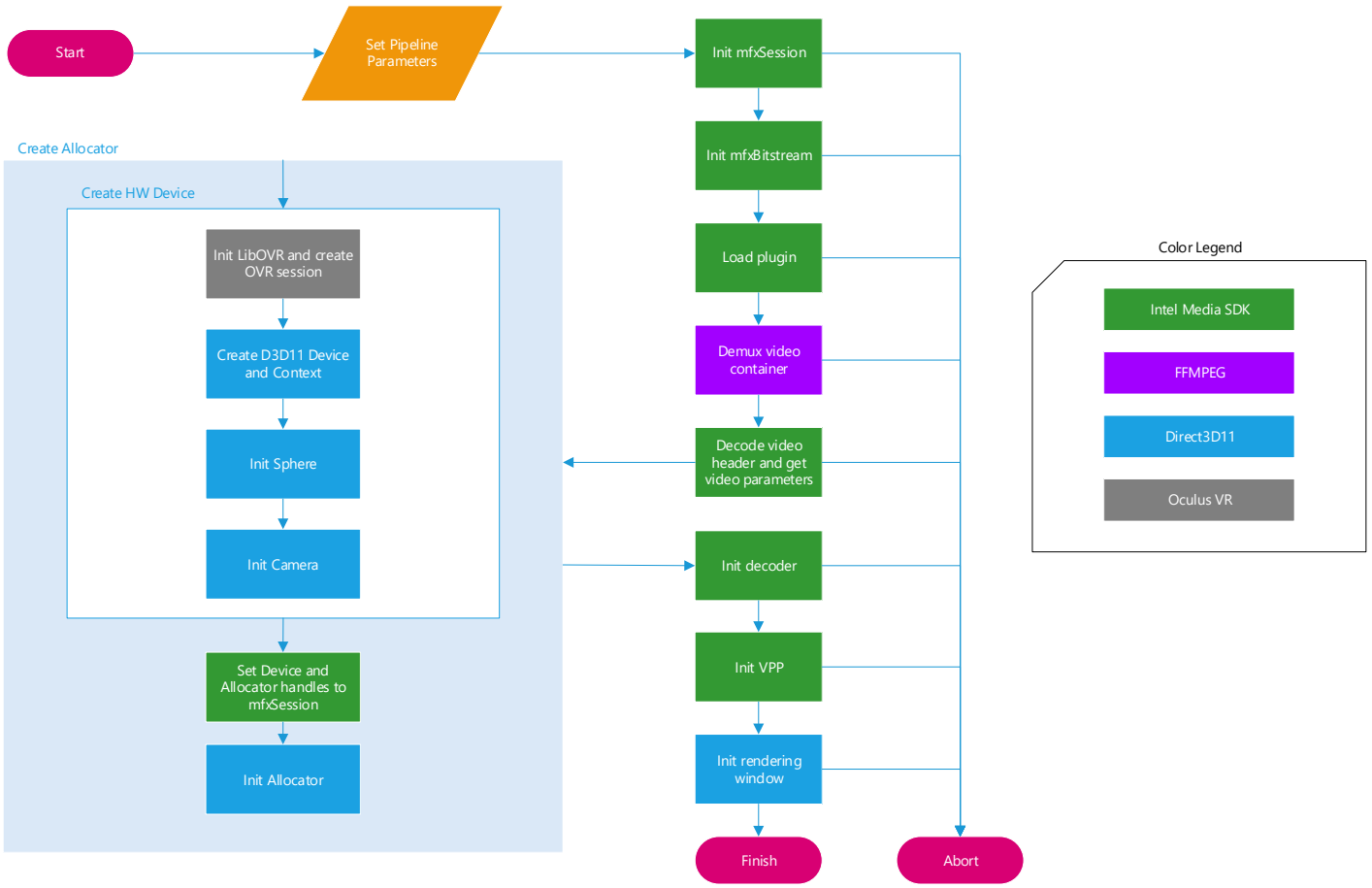
Belongs to sample_common. This class is responsible for creating and managing the on-screen mirror window. It is also responsible for handling window events and messages such as DESTROY, MAXIMIZE, KEYUP and KEYDOWN messages. All the KEYUP/KEYDOWN events are forwarded to the CD3DDeviceOVR to be handled by its OnKey() method.

CDecodingPipeline

Belongs to VR360Player. This class manages the decoding pipeline. It has two main methods which take care of the whole decode-render process. Init() is responsible of initializing and allocating all of the inner resources needed for the decoding session. After Init has successfully finished, Run() begins the decode-render loop, delivering the decoded frames to a different thread, thus efficiently implementing a producer-consumer approach.

CDecodingPipeline::Init()

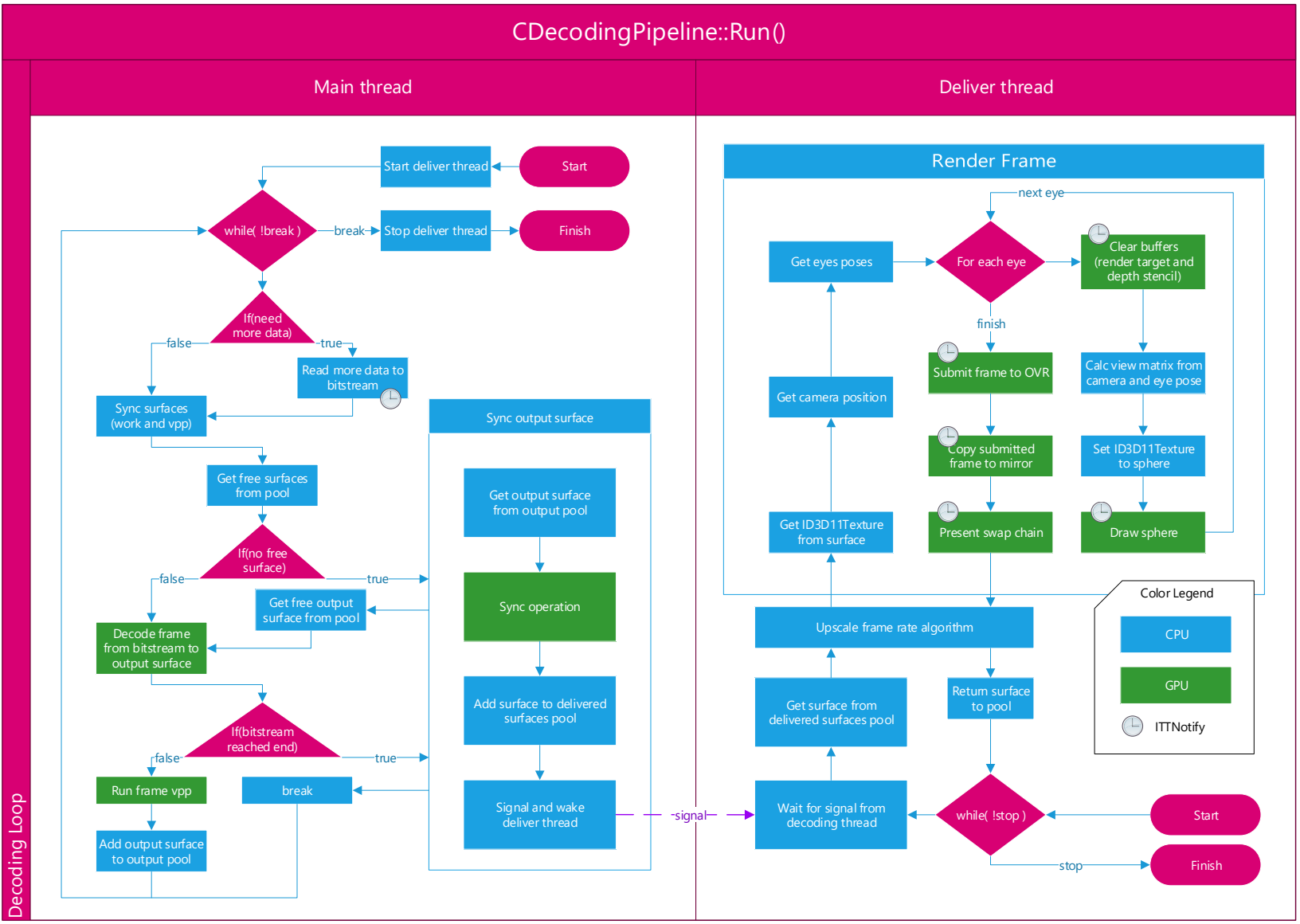
The following diagram shows the initialization process is more detail:



Each step in the initialization process is associated with a color that represents the library/API used. Any error along the initialization leads to abort.

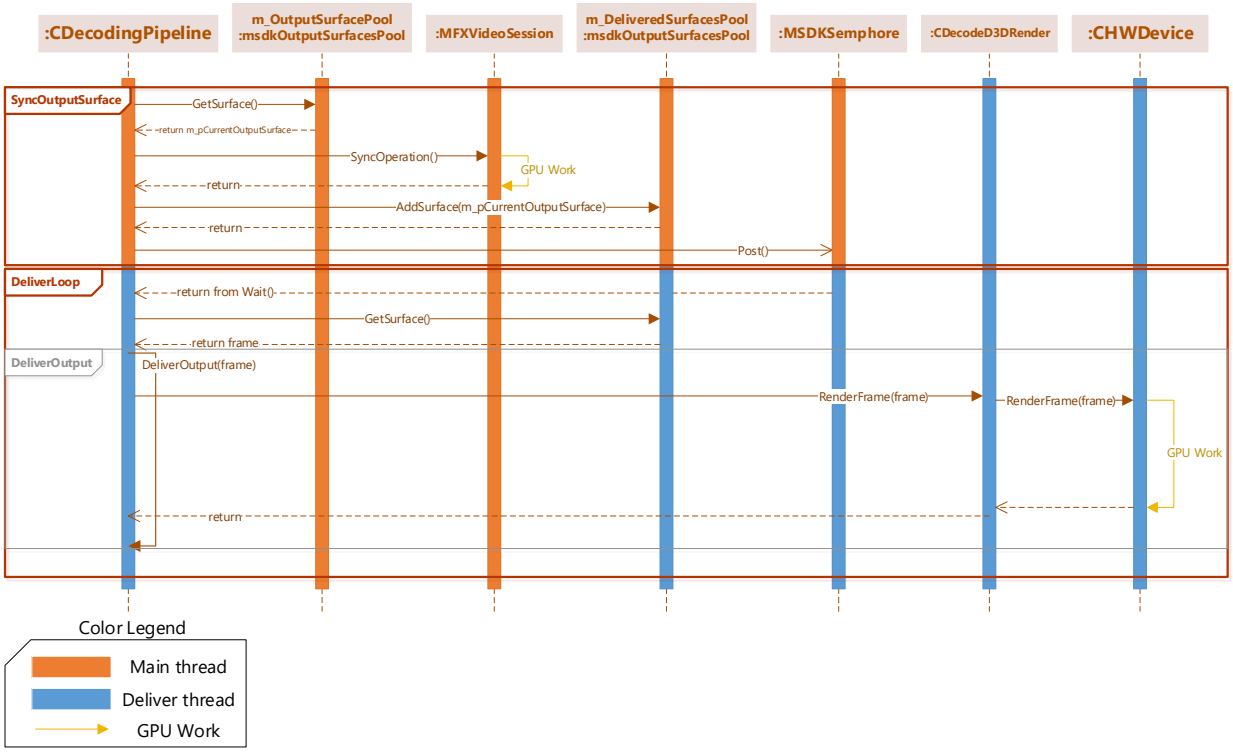
CDecodingPipeline::Run()

The following diagram shows the decode-render loop in more detail:



Actions that run on the CPU are colored in blue, while green represent procedures that run on the GPU.

A sequence diagram is given for the SyncOutputSurface() method:



Class Diagram

The relations between the classes descripted above are depicted in the following class diagram.